Filozofická fakulta Univerzity Karlovy v Praze

Katedra logiky

Logika dokazatelnosti a její filozofická reflexe

Logic of Provability from the Philosophical Point of View

Jan Filippi

Diplomová práce na oboru Logika

Vedoucí práce: doc. PhDr. Petr Jirků, Csc.

PRAHA 2008

Prohlašuji, že jsem tuto práci vypracoval samostatně a použil výhradně citovaných pramenů.
Souhlasím s půjčováním, kopírováním a zveřejňováním této práce.

v Praze dne 31. 8. 2008

Abstrakt

V matematické části je zpracováno téma autoreference v aritmetice. Při úvahách i důkazech je užit vyšší programovací jazyk, což umožňuje dospět ke známým výsledkům Gödela, Rossera a Löba přirozeným způsobem. V závěru je formulován další podobný problém a návrhy jeho řešení.
Ve filozofické části jsou diskutovány paralely výsledků logiky dokazatelnosti v humanitních vědách. Je zkoumán rozpor mezi determinismem a existencí svobodné vůle a co k tomuto rozporu může říci teorie algoritmů.

Abstract

The mathematical part of this thesis studies the phenomenon of self-reference in arithmetics. A higher programming language is used to present ideas and proofs, enabling us to reach the known results of Gödel, Rosser and Löb in a natural way. At the end of this part, we formulate another problem of a similar type and propose some ways to solve it.
The philosophical part draws analogies between statements originating in the logic of provability and their application in humanities. We study the clash between determinism and existence of free will, and apply the theory of recursive functions to this issue.

Obsah

1 Úvod **5**

2 Přípravné úvahy **8**

3 Formální aparát **11**

4 Aplikace – první část **25**

5 Věta o rekurzi **29**

6 Aplikace – druhá část **41**

7 Löbův důkaz **47**

8 Poslední na řadě **50**

9 Reflexe ve společenských vědách **54**

Text může být buď přesný
nebo pochopitelný,
nikoliv však oboje současně.
Leonhard Euler

1 Úvod

Tato práce je příběhem. Příběh začal na střední škole v hodinách matematiky.

V osnovách bylo zahrnuto i téma logika, do příběhu ale nepatří. Určité místo bylo věnováno matematickým důkazům se zvláštní důrazem na indukci pro přirozená čísla.

Z toho ve mně zůstaly dva dojmy.

Prvním byl tehdy jen intuitivně vycítěný rozdíl mezi deterministicky a nedeterministicky polynomiální složitostí problémů. Už tehdy byl většině z nás jasný onen propastný rozdíl mezi schopností ověřit správnost důkazu a schopností na takový důkaz přijít.

Druhý dojem se týkal hledání důkazů. Při řešení cvičení typu „dokažte, že pro každé přirozené číslo k existují prvočísla p a q taková, že $q-p > k$ a mezi p a q žádná prvočísla nejsou“ mě se stoupající obtížností úloh napadlo, že třeba některé důkazy ani nelze nalézt, i když protipříklad také ne.

V dnes mně blízké logické terminologii to mohu formulovat, že předpoklady, ze kterých dokazujeme, možná netvoří úplný systém.

Pamatuji se, že jsem tuto otázku tehdy položil učitelce spolu s další otázkou, co kdybychom pouze dokázali, že daný dů-

kaz existuje. Její odpověď na první část je lepší nezmiňovat (něco v tom smyslu, že hodně záleží na kvalitě toho kterého matematika, když je to špičkový matematik, znamená to, že důkaz neexistuje) zato odpověď na druhou otázku byla inspirativnější. Pravila, že dokázat, že existuje důkaz, je to samé jako to přímo dokázat.

Další setkání s tímto problémem se odehrálo pri přednášce „Metamatematika teorií množin“, kdy přednášející A. Sochor dokazoval, že Gödel-Bernaysova teorie množin je ekvikonzistentní s teorií Zermelo-Fraenkelovou.

První to dokázala I. L. Novak [14] v roce 1950 využitím teorie modelů (tento důkaz je také v Sochorově knize [20]). V roce 1950 pak totéž dokázal J. R. Shoenfield finitistickou metodou [17]. To znamená, že podal jasný algoritmus jak důkaz libovolné množinové sentence z axiomů GB předělat na důkaz této sentence z axiomů ZF.

Jaký je princip demonstrace metodou přes modely a proč k tomu tenkrát Sochor dodal, že obecně se považuje Shoenfieldův důkaz za hodnotnější (a ten také na přednášce předvedl)?

V rychlosti se dá důkaz přes modely popsat takto:

– φ buď dokazatelná v GB
– tedy platí v každém modelu GB
– z toho dokážeme (to je jádro práce), že platí v každém modelu ZF
– vzhledem k větě o úplnosti je φ dokazatelná v ZF

Když si postup důkazu rozmyslíme, vidíme, že z důkazu φ z axiomů ZF jsme dostali existenci důkazu φ z aximů GB.

V naší metateorii (což je vzhledem k nutnosti formulace věty o úplnosti nějaká teorie množin) jsme dokázali existenci důkazu. Nemáme ale žádný návod, jak tento důkaz sepsat (ten podal až Shoenfield).

A jsme znovu zpátky u odpovědi paní učitelky.

Co nám říká existence důkazu formule φ? Přesněji, co máme zaručeno, jestliže dokážeme, že existuje důkaz φ?

Pokud to znamená, že je formule φ dokazatelná, existuje obecná metoda, jak tento její důkaz získat?

A pokud ne, mohlo by se nám stát, že dokážeme $\neg\varphi$? Bylo by to ekvivalentní nalezení důkazu sporu?

K těmto otázkám má hodně co říci logika dokazatelnosti, respektive idea, která stála u jejího zrodu. Ukazuje se, že kdykoli napíšeme formuli φ a důkaz (v aritmetice), že existence důkazu formule φ implikuje samotnou φ, znamená to, že jsme schopni tento důkaz transformovat přímo na důkaz φ.

Podrobnější úvahou o formalizaci metajazyka v aritmetice nakonec dostaneme, že předpoklad existence výše zmíněné obecné metody vede k existenci důkazu sporu (pozor ne k důkazu sporu, jen k jeho existenci).

Úvahy jsou to zapeklité a rozum se při jejich sledování velmi lehko nechá svést na zcestí.

Práce je mimo jiné příběhem hledání cesty k odpovědím na tyto otázky.

2 Přípravné úvahy

Hlavním nástrojem textu je formulace problémů a postupů vedoucích přes Gödelovy věty až k logice dokazatelnosti řečí teorie rekurze s použitím vyššího programovacího jazyka (co možná neformálně ale bez nepřesností) namísto běžně užívané formalizace v prvořádovém aritmetickém jazyce.

Jaké přístupy máme k dispozici? Nabízí se použít vícesortovou logiku, kde kromě objektů – čísel budeme mít ještě objekty – programy. To by mělo výhodu, že by byly přípustné formule typu „existuje program těch a těch vlastností“. Na druhou stranu, naším jediným cílem je metodologické zjednodušení, touto cestou se proto pouštět nebudeme. Programy pro nás zůstanou na metamatematické hladině, podobně jako formule nebo termy aritmetického jazyka.

Úvahy o tvrzeních získaných diagonalizací (specielně o nerozhodnutelných tvrzeních) mají dva aspekty. První je principiální a související úvahy nejsou podle mého názoru obtížné (alespoň k pochopení). Druhým aspektem je technická stránka věci. Ve všech textech, které jsem o tom kdy četl, bylo pochopení technické části nezbytné pro přesné chápání principiální části. Je mojí neskromnou snahou pokusit se tyto dvě části oddělit a čtenáři, kterým může být nematematicky vzdělaný filozof nebo humanitní vědec ale třeba i fyzik, který matematiku zná, ale nikdy ji nepoužíval takovýmto způsobem, umožnit při přeskakování technikalit přesně pochopit, co přeskakuje. Z tohoto důvodu volím následující poněkud netradiční postup.

Jazykem naší teorie bude aritmetický jazyk plus programy

ve vyšším programovacím jazyce. Běžným způsobem nadefinujeme aritmetický term, aritmetickou formuli a sepíšeme axiomy.

Kromě toho zadefinujeme (přesněji metadefinujeme), co to jsou dobře sepsané programy v jazyce Pascal, o které nám také – kromě přirozených čísel – půjde (přijmeme jistá nepodstatná omezení). To můžeme, pro Pascal je tak jako pro všechny programovací jazyky k dispozici tzv. syntaktický diagram, což je vlastně ona metadefinice. Naším hlavní zájmem zůstanou přirozená čísla. Budeme ale chtít mít možnost hovořit o tom, že nějaký program se na vstupních datech zastaví po určitém počtu kroků, že existuje určitý počet kroků, po kterém výpočet stojí, a podobně. Co nebudeme potřebovat (na matematické hladině) je formulace otázek o existenci programů určitých vlastností.

Předvedeme, jak všechna naše tvrzení přeložit do čistě aritmetického jazyka. Tím ukážeme, že z formálního hlediska je lhostejné, jakou teorii používáme. Sice nic nezískáme, ale také nic neztratíme.

Výhodou tohoto přístupu je obrovská svoboda uvažování a vyjadřování. Budeme si moci bez obav dovolit úvahy o programech, které hledají důkaz nějakého tvrzení, třeba i důkaz tvrzení, že se nějaký jiný program zastaví či nezastaví. A také uvidíme fascinující větu o rekurzi, která je v řeči teorie algoritmů intelektuálním skvostem, zatímco Gödelovo diagonální lemma, které jí v aritmetice odpovídá, se při čtení zdá být nudným technickým nástrojem navíc získaným trikem.

Je třeba říci, že tato práce již byla historicky několikrát udělána.

Gödela nebudu počítat, neboť on se držel aritmetického přístupu. S jistým nadhledem lze říci, že i jeho práce má cosi společného s programováním. Velmi se totiž podobá práci s jazykem LISP. V tomto jazyce je všechno funkce a neexistují jiné objekty. Má velice blízko k lambda kalkulu.

Právě lambda klakul naformuloval k těmto účelům další logik, který pracoval v této oblasti, S. C. Kleene.

Třetím člověkem, kterého je určitě třeba zmínit, je Alan Turing, který použil z dnešního pohledu assembler, což už má velice blízko k procedurálnímu programovacímu jazyku současného typu. Doba tím byla těhotná a stejné či podobné myšlenky paralelně a nezávisle vznikaly v myslích více matematiků (např. také Emil Post a tzv. ruská škola konstruktivismu A. A. Markova).

3 Formální aparát

Plán této kapitoly:
Předložit formální teorii, ve které budeme moci přirozeným způsobem hovořit o číslech. K tomu chceme kromě formulí užívat i programy zpracovávající texty, tzn. zpracovávající i formule, důkazy a zápisy programů. Ukázat, že naše teorie je v podstatě konzervativním rozšířením Peanovy aritmetiky o definice a že tedy jakékoli tvrzení v ní formulované jsme schopni přeložit do čistě aritmetického jazyka.

Syntax:

Vytváříme rozšíření PA, vezmeme její nejčastěji užívaný jazyk $\{+, \cdot, 0, S, \leq, <\}$ a k němu přidáme další symboly. Podotkněme, že ve výše zmíněném výčtu nejsou všechny symboly jazyka ale pouze mimologické. Z logických symbolů je v jazyku například také spočetně nekonečně mnoho symbolů pro proměnné.
Nyní chceme otevřít možnost mluvit o programech. Nabízí se přidat do abecedy všechny symboly z ASCII tabulky, přidat novou sortu proměnných, definovat programové termy (= programy). Tento přístup by měl výhodu, že by byl standardní, umožnil by nám přes programy kvantifikovat a tím vyjadřovat tvrzení o existenci či neexistenci programů s různými vlastnostmi. Důvodem proti je přílišná složitost tohoto přístupu a to, že chceme, aby všechno bylo konstruktivní. Netoužíme po tvrzeních, že nějaký program existuje, vždy ho budeme chtít mít napsaný. Proto nám budou postačovat programy na metamatematické hladině. Výsledek bude jed-

noduššší i když netradiční.

Do jazyka přidáme nekonečně mnoho symbolů, totiž všechny textové řetězce. Jedním jejich poddruhem budou syntakticky správně napsané programy v jazyce Pascal bez vstupů a výstupů. Druhým poddruhem budou programy bez výstupů s jedním vstupem. To, že máme spočetně nekonečný jazyk, není nic proti ničemu, viz předchozí poznámka o nekonečném množství proměnných. Ve všech úvahách a zápisech jich užijeme vždy jen konečně mnoho.

Přidáme nový binární predikát !, který bude pro program bez vstupu i výstupu a číslo vyjadřovat skutečnost, že daný program po tomto počtu kroků stojí.

Definice termů:

a) Všechny obvykle definované termy aritmetického jazyka jsou termy – budeme jim říkat termy *aritmetického druhu* nebo krátce *aritmetické termy.*

b) Je-li t jakýkoli řetězec symbolů abecedy (včetně logických symbolů), je termem – budeme říkat t-termem.

c) Je-li p syntakticky dobře napsaný program v jazyce Pascal nemající žádný vstupní ani výstupní soubor, je termem – budeme říkat 0-programem.

d) Je-li p syntakticky dobře napsaný program v jazyce Pascal mající jeden vstupní a žádný výstupní soubor, je termem – budeme říkat 1-programem.

Příklad:

Textový řetězec `program␣prik1;begin␣end.` je 0-program.

Textový řetězec `program␣prik2(input);begin␣end.` je 1-program. Oba jsou samozřejmě t-termy.

Definice formulí:

a) Všechny atomární formule čistě aritmetického jazyka jsou atomární formule.

b) Je-li p 0-program a k aritmetický term (například proměnná), je $!(p, k)$ atomární formule.

c) Jsou-li φ a ψ formule a $\bowtie$ logická spojka, je slovo $\varphi \bowtie \psi$ a slovo $\neg\varphi$ formule.

d) Je-li n proměnná (nemáme jiné proměnné než pro čísla) a φ formule, jsou slova $\exists n\ \varphi$ a $\forall n\ \varphi$ formule.

Poznámka: Vidíme, že máme možnost kvantifikovat přes přirozená čísla a to i tehdy, když se predikát ! vyskytuje uvnitř formule. Nemáme však možnost kvantifikovat přes programy, nemáme dokonce pro ně ani proměnné.

Definice: Zápisem $!p$ budeme značit formuli $\exists n\ !(p, n)$

Axiomy:

a) Všechny axiomy Peanovy aritmetiky jsou axiomy.

b) Je-li p 0-program a k aritmetický term bez proměnných (neboli takzvaný numerál) a program p se po spuštění zastaví dříve než za k kroků, je formule $!(p, k)$ axiomem.

c) Uvidíme později, možná ještě něco přibude

Odvozovací pravidla:

– modus ponens

– generalizace

Tuto naši teorii budem značit NT.

Poznámka:

a) Může se zdát divné, že to, zda je něco axiomem nebo ne, záleží na výsledku takovéhoto procesu. Skoro to vypadá, že o axiomech rozhodujeme fyzikálním experimentem. Vezmeme technickou realizaci univerzálního algoritmu (počítač), spustíme na něm program a stojíme u něj se stopkami v ruce a kontrolujeme, zda v okamžiku zapípání hodinek kontrolka svítí či nesvítí. Ale uvědomme si, že v tuto chvíli jsme na úrovni metamatematiky. Jak se tento návod, jak rozpoznat mezi hromadou formulí axiom, liší od návodu, jak mezi hromadou textových řetězců rozpoznat dobře utvořenou formuli? Nijak. Rozhodujeme o tom algoritmickým procesem na metamatematické hladině. Jediné, co požadujeme, aby byl tento proces rekurzivní, neboli algoritmizovatelný s jistotou zastavení a tím pádem i rozhodnutí.

b) Schema indukce z PA můžeme užívat i na nearitmetické formule, tj. na formule, ve kterých je použit predikát !

c) Co budeme ještě přidávat? A měli bychom vůbec něco přidávat? Zatím máme mezi axiomy spoustu formulí bez proměnných. Ale neumíme dokázat například takovouto formuli (p značí nějaký konkrétní 0-program):
$\forall n\, \forall k > n \;\; !(\mathsf{p}, n) \to !(\mathsf{p}, k)$. Jsme schopni to samozřejmě

dokázat pro každé jednotlivé numerály n a k. Abychom mohli použít axiom z PA o indukci, potřebovali bychom vědět, že $\forall n\ !(\boldsymbol{p}, n) \to !(\boldsymbol{p}, n+1)$.

d) Povšimněme si, že 1-programy ani t-termy se nevyskutují vůbec nikde. Ani v definici formulí a tím spíše ani v axiomech. Rovněž tak nemáme žádná odvozovací pravidla, která by nám z předpokladu $!(\boldsymbol{p}, n)$ odvodila formuli $!(\boldsymbol{q}, n)$ například tehdy, když se $\boldsymbol{q}$ liší od $\boldsymbol{p}$ jen nepodstatně. Další úvahy o tomto problému odložíme, zatím je jasné alespoň to, že naše axiomy nás nedovedou ke sporu.

Sémantika:

Univerzum modelu je složeno ze dvou množin. První je standardní množina přirozených čísel, druhou je množina všech t-termů. V ní jsou vyděleny dvě disjunktní podmnožiny, množina všech 0-programů a množina všech 1-programů. Kromě toho jsou v ní samozřejmě i zápisy všech možných programů a také všechny formule a všechny důkazy atd.
Všechny aritmetické funkční a predikátové symboly jsou realizovány pouze na první množině (na druhé a třetí je případně můžeme jakkoli dodefinovat) a to stejně jako ve standardním modelu.
Realizací predikátu ! je binární relace, která obsahuje dvojici ⟨0-program, číslo⟩ právě tehdy, pokud se program zastaví po nejvýše tolika krocích.

Metamatematická úvaha a konstrukce:

Programy jsme přidali mezi termy, abychom mohli o ! mluvit

jako o predikátu. Stejně tak dobře bychom mohli programy přidat jako další pojem na stejné úrovni s pojmem formule. Definice atomárních formulí obsahujících ! by pak ale nebyla přirozená.

Představme si, že máme na papíře vytisklý program p v Pascalu, který má jeden vstupní soubor (nazvaný standardně `input`) a žádný výstup – neboli náš 1-program. Na druhém papíře je vytisklý nějaký textový soubor. Cílem je pozměnit program p tak, aby výsledkem byl program bez vstupu – neboli 0-program – pracující evidentně stejně, jako by pracoval původně p na soubor vytisknutý na druhém papíře.
Nejprve se podíváme na druhý papír, kolik je na něm znaků, dejme tomu, že jich je 782. Program p pak pozměníme následovně:

1. Smažeme znaky `(input)` hned za jeho jménem, takže nebude mít vstup.
2. Do hlavičky přidáme deklarace, kterými vytvoříme místo pro vstup z druhého papíru (identifikátor `text`, říká, že proměnná `input` je typu textový soubor):
   ```
   const delkavstupu=782;
   type typvstupu=packed array[1..delkavstupu];
   var vstup:typvstupu;
       input:text;
   ```
3. V těle programu hned na začátku do proměnné `vstup` vložíme celý obsah druhého papíru příkazem:
   `vstup:='`*bla bla opisujeme druhý papír*`';`
4. Naplníme (teď je to pracovní nikoli vstupní) soubor `input` znaky z řetězce `vstup`:
   ```
   rewrite(input);
   for i:=1 to delkavstupu do
   ```

```
begin
  write(input,vstup[i])
end;
```

Tím je práce hotova, máme pozměněný 1-program p, je z něho teď 0-program.

Poznámka: Možné zádrhele a komplikace

– předpokládali jsme, že námi použité identifikátory se v původním programu nevyskytují
– program na druhém papíře bereme jako textový soubor s mezerami ale bez konců řádek

To jsou nepodstatné drobnosti. Co je mírně zajímavější komplikací je řešení zápisu apostrofů (znak '). V Pascalu platí (aby se odlišila funkce apostrofu v syntaxi jazyka jakožto vyznačovače textových řetězců od apostrofu jako pouhého symbolu), že pro jeho zápis ho musíme zdvojit. Takže ve skutečnosti musíme být obšírnější. Při opisování papíru do proměnné `vstup` musíme místo každého apostrofu napsat dva – ale délku řetězce tím neprodloužíme, protože překladač při čtení nahradí dva po sobě jdoucí apostrofy jedním. Při zápisu do souboru už k žádné podobné komplikaci nedochází. Může se to zdát nepodstatné, ale na tento problém se naráží doslova všude. Samozřejmě při psaní Unix nebo Windows scriptů pro práci s textem ale i při filozofických úvahách o smyslu a významu. Uvidíme to na příkladu programu, který tiskne sám sebe.

Protože jinak v programu neděláme vůbec žádné změny, dovolujeme si prohlásit, že je evidentní, že takto získaný bezvstupový program pracuje navlas stejně jako původní na vstup

na druhém papíře.

Tímto způsobem dojdou užitku i ostatní termy našeho jazyka – t-termy a 1-programy. Když je budeme potřebovat, dosadíme je do nějakého konkrétního 1-programu, tím dostaneme 0-program, a z něho už můžeme tvořit formule.

Definice: Je-li p 1-program a d jakýkoliv t-term, zápisem $p(d)$ budeme značit 0-program vzniklý takovýmto dosazením d do programu p.

Následující úvaha slouží k vyjasnění některých pojmů a postupů.

Minulá definice označuje term, který je výsledkem určitého finitního procesu na jiné dva termy. Je to typická metadefinice. V běžné matematické teorii máme objekty teorie (v teorii množin množiny, v aritmetice čísla apod.) a dále syntaktické objekty, kterými o nich mluvíme – slova, termy, formule, důkazy. Jestliže je ale naším záměrem zkoumání samotného jazyka a jeho syntaxe, může velmi často dojít k nedorozumění a zmatení pojmů.

První možností pro takový záměr je vzít teorii, která je dostatečně silná na to, aby v ní šla celá syntax namodelovat. Klasická Gödelova práce spočívala právě v namodelování prvořádového jazyka v teorii přirozených čísel. Například nadefinoval predikát, který říká, že číslo je kódem formule atd. Má to tu výhodu, že ke zmatení dochází mnohem obtížněji, nevýhodou je těžkopádnost vyjadřování.

Druhou možností je vzít za objekty teorie přímo textové řetězce. Veškeré definice takovýchto predikátů v podstatě odpa-

dají respektive se trivializují. Co je ale velkou nevýhodou je možnost zmatení. Další nevýhodou je, že abychom ukázali, že můžeme přenést takto dosažené výsledky (například o dedukтivní nezúplnitelnosti) do teorie, která nás zajímá především (do Peanovy aritmetiky nebo teorie množin), budeme stejně muset nakonec tu práci s překlady vykonat. Nejlepším přístupem se mi proto jeví rozvíjet oboje souběžně.

Neustále je ale potřeba bezpodmínečně rozlišovat hladinu matematickou od hladiny metamatematické. Je lehké napsat na papír zápis „Tato sentence je nedokazatelná.“ Problémem u takovéhoto zápisu je ukázat, že se skutečně jedná o formuli našeho jazyka (v tomto případě se nám to nepovede, prostě to žádná formule není). Teprve po spoustě práce jsme schopni napsat konkrétní formuli a o té prohlásit, že má takovouto interpretaci.

Důslednou cestou pro druhou možnost je λ-kalkul případně programovací jazyk LISP, který z něho vychází. Nejsou v něm jiné objekty než seznamy (textové řetězce) a vše je pouze jejich manipulací. Věta o rekurzi, kterou v dalším textu uvidíme, je v podání λ-kalkulu tak přirozená, že by napadla doslova každého.

Vraťme se ale k našemu přístupu. p značí jednovstupový program. Zápisem $p(d)$ jsme označili nulavstupový program, který vznikne „dosazením“ řetězce d do programu p.

Pozor zápis NEZNAČÍ výsledek spuštění programu p na data d. Tady je velký zdroj možných nedorozumění, kterým se ale dá těžko zabránit. Matematici a fyzici zažívají stejný druh zmatení. Fyzik například typicky definuje funkci zápisem $y = f(x)$. Co teď znamená, když napíše $f(x)$? Myslí tím celou

funkci nebo nějakou konkrétní hodnotu pro konkrétní x? A ještě něco. Předveďme jeden takový myšlenkový pochod fyzika – definujme funkci f zápisem $y = x^2 + x + 1$. Co teď znamená, když napíšeme $f(4)$? Je několik možností:

a) Číslo 21, které dostaneme tak, že vezmeme celou funkci coby množinu uspořádaných dvojic (to jsme se dozvěděli v hodinách matematiky, že je funkce) a z grafu odečteme hodnotu y pro $x = 4$.
b) Výraz $4^2 + 4 + 1$, který dostaneme dosazením konstanty 4 za x do výrazu definujícího funkci f.
c) Číslo 21, které dostaneme vyčíslením předchozího výrazu.

Naší definici má nejblíž případ *b*). Proč volíme tento význam pro slovo $p(d)$? Proč rovnou nespustíme program a nevezmeme jeho výsledek? Je k tomu několik důvodů.

Pozměňme definici f v předchozím příkladu na $f(x) = 1/(x-4)$ a znovu se zamysleme, co může znamenat výraz $f(4)$. Přístupy *a*) a *c*) tentokrát použít nemůžeme. Pamatujeme se jistě ze základní školy, že nulou se dělit nesmí, takže se o to raději ani nepokoušíme. Kdybychom se o to chtěli pokoušet, nejjednodušší algoritmus pro dělení vzatý přímo z definice je postupně zkoušet všechna čísla násobit dělitelem a kontrolovat, jestli se výsledek náhodou nerovná dělenému číslu. Pokud ano, držíme ho v ruce. Algoritmus skončí pro každý vstup – kromě případu, že se snažíme dělit nulou.

To je jeden důvod.

Druhým důvodem může být, že vyčíslením výrazu nutně ztrácíme určitou informaci (známé příklady, že výrok „Pavel neví, že $7 + 5 = 12$“ přejde vyčíslením na výrok „Pavel neví, že $12 = 12$“, což není totéž).

Na druhou stranu, chceme mít možnost říci o dvou programech, že jsou ekvivalentní ve smyslu, že počítají totéž. Volně řečeno, pokud dva programy bez vstupu skončí a vydají výsledek, je nám lhostejné, jestli k němu došly stejným způsobem, pro nás jsou ekvivalentní. Jsou navíc ekvivalentní s kratičkým programem, který prostě jen tiskne onen výsledek.

Pokud oba nebo jeden z nich ne a ne skončit, situace už není tak jednoduchá. Jaké jsou možnosti.

Jeden skončí, vydá výsledek, u druhého se nám nepoštěstilo. Ani dočkat se jeho zástavy, ani ji dokázat. V tom případě nemůžeme říci nic.

Třetí možnost je, že jeden skončí, vydá výsledek a u druhého se nám jednak podaří dokázat, že skončí, jednak, že když skončí, tak určitě s tím samým výsledkem[1]. Můžeme je prohlásit za ekvivalentní?

Čtvrtou možností je, že ani jeden nekončí a nedaří se nám ani uspět s dokazováním jejich nezástavy.

Pátou možností je, že se nám u obou podaří dokázat jejich nezástavu.

Zásadní pro tyto úvahy je, z jakých axiomů a jakými pravidly tyto důkazy získáváme. A nemyslíme tím jen axiomy teorie. Je možno uvažovat i jiné logické systémy – jmenovitě intuicionistickou logiku. Přiznávám, že tuto problematiku nemám rozmyšlenu, ale tipnul bych si, že při budování aritmetiky nad intuicionistickou logikou by takové problémy nevznikaly. Důvod spatřuji v tom, že při použití intuicionistické logiky je důkaz existence popisem konstrukce.

[1] zase jsme u odpovědi paní učitelky

Rozumím tomu tak, že pokud paní učitelka měla namysli důkazy v intuicionistické logice, je její odpověď správná.

Konzervativnost:

Nyní ukážeme, že naši teorii lze chápat podobně jako konzervativní rozšiřování teorií o definice, že naše obohacení jazyka i axiomů je z formálního pohledu zbytečné. Tato část je právě onou technicky obtížnou prací, kterou je potřeba vykonat k dosažení vět o neúplnosti i vět logiky dokazatelnosti. Postup jen nastíníme. Podotkněme, že jde o něco mírně odlišného od metavět například v Sochorově knize [19], ale ne principielně.

Půjde nám pouze o překlad formulí. Nebudeme definovat překlady programových termů. Protože nemáme proměnné pro programy (tím pádem přes ně nemůžeme kvantifikovat), bude stačit metamatematicky ukázat překlad každé atomární formule $!(p, t)$, kde symbolem p rozumíme konkrétní 0-program neboli konkrétní program v Pascalu bez vstupů a výstupů a symbolem t aritmetický term, tzn. například proměnnou nebo numerál (možností je víc). Překlad budeme značit opruhováním $\overline{!(p, t)}$.

Stačí uvážit situaci, že t je proměnná, třeba k. V případě, že t není proměnná, přeložíme formuli s nějakou pomocnou proměnnou místo t a po překladu ji všude nahradíme termem t. Chceme, aby $\overline{!(p, k)}$ byla aritmetická formule s jedinou volnou proměnnou k a aby pro každý numerál n bylo $\mathsf{NT} \vdash !(p, n)$ právě tehdy, když $\mathsf{PA} \vdash \overline{!(p, n)}$. Podrobně to dělat nebudeme, uvedeme jen několik poznámek:

a) Nestačí mít postup, jak toto napsat pro každý p a pro

každý numerál n. Potřebujeme místo n mít proměnnou, budeme přes ni totiž chtít v dalších úvahách kvantifikovat.

b) Nástin konstrukce: Nejjednodušší je využít konstrukce univerzálního stroje – programu neboli interpreteru. Dostane nulavstupový program p a term t. Term t nejprve vyčíslí a pak nechá p běžet tento počet kroků. Po té se podívá, jestli emulovaný program ještě běží nebo ne, a podle toho rozhodne. Z této úvahy plyne, že stačí provést požadovaný překlad jen pro jeden konkrétní program – interpreter. Je to právě ta technická část, kterou si můžeme dovolit přeskočit, pokud alespoň tušíme, jak by se to technicky udělalo.

c) Z předchozího by mělo být jasné, že získaná formule sice může obsahovat kvantifikátory, ale jen omezené. Jako mez může ovšem být užita i volná proměnná. Formule φ je tedy Δ_0.

d) Celou konstrukci je možné algoritmizovat. Můžeme předpokládat, že máme program, který to vykonává. Jako vstup dostane program bez vstupu a numerál (to je metamatematický objekt neboli textový řetězec). Vždy skončí a výstupem bude textový řetězec - zápis Δ_0 formule v jazyce PA.

Tím jsme ukázali, že každou formuli naší teorie umíme přeložit do čistě aritmetického jazyka. Ještě je potřeba se zamyslet, jestli po překladu platí i všechny naše axiomy. Překladem všech axiomů jsou Δ_0 sentence, které jsou v PA dokazatelné – to říká mimo jiné věta o Σ_1 úplnosti. To ale nemusíme ukazovat, klidně bychom je mohli všechny do PA přidat.

Ještě se zamysleme nad opačným postupem. Jestli totiž nelze

pro každou Δ_0 formuli $\varphi(n)$ aritmetického jazyka s jednou volnou proměnnou mechanicky sepsat 0-program p, že když přeložíme zase zpátky do aritmetického jazyka formuli $!(\mathsf{p}, n)$ na $\overline{!(p, n)}$ dostaneme původní formuli.

Skutečně to takto funguje, jen můžeme dostat jinou formuli ovšem dokazatelně ekvivalentní s původní.

Jakmile toto máme, je zřejmé, co potřebujeme doplnit na ono prázdné místo ve výčtu axiomů. Pro každý programový term p a pro každý aritmetický term t (i proměnnou, to je důležité) přidáme axiom $!(\mathsf{p}, \mathsf{t}) \equiv \overline{!(\mathsf{p}, \mathsf{t})}$. Z toho dostaneme, že formule tvaru $!(\mathsf{p}, \mathsf{t})$ jsou právě Δ_0 formule aritmetiky a z toho, že formule $!\mathsf{p}$ jsou právě Σ_1 formule aritmetiky.

4 Aplikace – první část

Vstupními soubory programů mohou být textové soubory (a v našem případě nic jiného). Zápisy programů jsou také textové soubory. Od dob Cantora a jeho objevu diagonální metody je proto přímo povinností uvažovat o spouštění programů na zápisy jiných programů nebo na svůj vlastní zápis. Úvodním bude samozřejmě Turingův příklad halting problému, že totiž neexistuje program, který by rozhodoval zastavovaní jiných programů. Při technickém provedení diagonalizace oceníme ony metamatematické úvahy o dosazování vstupu. Potřebujeme totiž mít možnost zmenšit počet vstupů. Lépe to bude vidět konkrétně.

Předpokládejme, že nám někdo přinese program, o kterém prohlašuje, že rozhoduje zástavu každého programu na jakákoliv vstupní data. Program má evidentně dva vstupy, na první náleží zápis programu, který nás zajímá, na druhé jeho vstupní data. Naším úkolem je přesvědčit příchozího, že nemůže mít pravdu.

Nejprve upravíme rozhodovací program tak, aby měl vstup pouze jeden a kontroloval, jestli se program na vstupu zastaví na svůj vlastní zápis. To je triviální záležitost a pokud přinesený program dělá, co má, bude fungovat i tento upravený. Dále ho upravme tak, že na jeho konec přidáme instrukce, které zařídí, že pokud zjistí, že se vstupní program na svůj vlastní zápis zastaví, skočí do nekonečné smyčky, a pokud zjistí, že se vstupní program na svůj zápis nezastaví, neudělá nic a skončí.

Úvaha, co se stane, když tento finální produkt spustíme na

vlastní zápis vede v obou případech ke sporu, z čehož plyne, že takový program nemůže existovat.

Kde jsme použili ono dosazování? Zatím nikde, protože jsme nekonstruovali žádnou formuli. Celá úvaha byla metamatematická a jen ukázala, že se nám nepodaří napsat program určitých vlastností.

Při konstrukci následujícího programu, který označíme g a nazveme Gödelovým, již ono dosazení použijeme. Nejprve napišme 1-program s. Vstupní soubor se předpokládá být taktéž zápisem 1-programu a označíme ho p. Popišme nyní neformálně ale přesně, jak napsat program s. Vezme vstup p a zkontroluje, je-li zápisem jednovstupového programu. Pokud ne, končí. Pokud ano, dosadí ho do něho samého. Tím získá zápis řekněme programu $u = p(p)$. Při dosazování p do p postupuje přesně tak, jak jsme popsali v předchozím – mírně p upraví a na začátek přidá celý zápis jeho původní podoby. Hlavní je, že je jisté, že vzniklý program u, který teď držíme v paměti, nemá žádný vstup a pracuje přesně stejně jako p spuštěný na sebe sama. Nyní vyrobíme aritmetickou formuli $!u$ a z ní hned $\neg!u$, která má význam „u nikdy neskončí“ neboli „p spuštěn na svůj vlastní zápis nikdy neskončí“. Všechny dosavadní kroky jsou vykonávány velmi rychle a s jistotou, že obdržíme výsledek. Následuje hledání důkazu této posledně uvedené formule z axiomů PA. Pokud takový důkaz nalezne, program s končí.

Ještě upřesněme, v čem spočívá hledání důkazu nějaké formule. Algoritmus prochází postupně všechny textové řetězce (to lze, kratší řetězec před delším řetězcem a řetězce stejné délky lexikograficky). Pro každý řetězec kontroluje, je-li důkazem, a pokud ano, jestli dokazuje tu formuli.

Teď diagonalizace: dosaďme s do s. Formálně $\mathsf{g}{:=}\mathsf{s}(\mathsf{s})$

Při konstrukci programu g jsme nepoužili žádný proces, který by mohl neskončit. Vše je finitní. Vstupními parametry jsou jazyk a axiomy teorie (v našem případě Peanovy aritmetiky) a jedna fixní metoda překladu formule $!p$ do aritmetického jazyka. V lidovém žargonu pro programy bychom řekli, že program g je program-sabotér. „On teda jako pracuje, ale nedělá nic jiného, než že hledá důvody, proč stejně nikdy neskončí.“ Mimochodem, můžeme napsat program, který nejdříve bude vykonávat něco užitečného (třeba hledat důkaz Goldbachovy hypotézy), a po případném ukončení této činnosti pokračovat hledáním důkazu své nezástavy. Pro takovýto program – maskovaného sabotéra – platí naprosto stejné úvahy jako pro g.

Diagonalizace nám opravňuje správnost následujících úvah.

Program g se nemůže zastavit, kdyby se zastavil, znamenalo by to, že našel důkaz toho, že se nezastaví, tzn. měli bychom důkaz sporu v PA. Současně se nám ale skutečnost, že se nezastaví, nemůže v PA podařit dokázat. Kdyby se nám to totiž povedlo, on by na ten důkaz přišel také a zastavil se. Ještě jednou přesná metaúvaha: kdybychom napsali na papír důkaz jeho nezástavy, pak bychom ho mohli spustit, sledovat krok po kroku, být nakonec svědky jeho zastavení (v momentě kdy nakonec přijde na tento důkaz o své nezástavě) a z toho všeho sepsat důkaz o jeho zástavě. Neboli na základě důkazu o jeho nezástavě jsme schopni sepsat důkaz o jeho zástavě. Neboli na základě důkazu o jeho nezástavě jsme schopni sepsat důkaz sporu v PA. Tato část úvahy tedy stojí na předpokladu bezespornosti PA.

Naopak představme si, že máme na papíře zapsaný důkaz o jeho zástavě. Náhle jsme zpátky u otázky zmíněné v samém úvodu textu a přemýšlíme, jestli tenkrát paní učitelka měla pravdu. Ano nebo ne? Můžeme z předpokladu, že máme v ruce důkaz o zástavě programu, který něco hledá, vyvodit, že to něco máme, a dále s tím zacházet v úvahách, jako bychom to skutečně drželi? A pokud ano, v jakém smyslu a jak to používat?

Odborně řečeno, abychom se v úvaze pohli dopředu, musíme použít Σ_1 korektnost PA.

Podrobněji: máme na papíře napsaný důkaz tvrzení $!p$, což je Σ_1 formule. Rádi bychom z něho nějak získali to konkrétní číslo, jehož existence je postulována, v našem případě počet kroků výpočtu, po kterém program p už určitě stojí. Jakmile ho budeme mít, není už problém získat i to, co program hledá. Σ_1 korektnost nám říká, že takové číslo existuje, ale nedává žádný odhad na jeho velikost. V dalším uvidíme, že to je neodstranitelná vada. Pokračujme. Máme tedy počet kroků, po kterém program jistě stojí. Z jeho konstrukce je jasné, že se mohl zastavit jedině nalezením důkazu o své nezástavě. Spor.

Tato část úvahy stojí na předpokladu Σ_1 korektnosti, což je poměrě zvláštní vlastnost teorií. V dalším uvidíme, že ji nelze formalizovat, což je jistě argumentem pro hledání jiné formule, jejíž nerozhodnutelnost by se bez tohoto předpokladu obešla.

Dalších zajímavé programy předvedeme až po následující kapitole.

5 Věta o rekurzi

Přistupme nyní k formulaci a důkazu zásadního tvrzení našeho textu. Bude mu předcházet několik úvah.

Za prvé uvažujme nad negativním řešením halting problému. Jeho myšlenkou je diagonalizace a výsledkem je, že něco nemůže existovat. Překvapení z myšlenkového výkonu se odehrálo již před mnoha lety při studiu Cantorovy věty o nespočetnosti množiny reálných čísel a tento výsledek můžeme považovat jen za další za aplikaci.

Cantorova metoda nám obecně vzato zakazuje existenci příliš univerzálních objektů. V případě programů nám zakazuje existenci univerzálního rozhodovacího algoritmu. Podobně se středověcí scholastici rozhodli omezit boha rozumem, neboť všemocnost sama o sobě je sporná (ani křesťanský bůh nemůže stvořit tak těžký kámen, že ho sám nezvedne).

V souladu s tím se zdá, že by neměl existovat univerzální program. Program, který na vstup dostane zápis jiného programu a pracuje podle něj, to znamená jako on. Pokud jsme někdy programovali, mělo by nám být zřejmé, že napsat takový program zase není až tak těžké (takový program se nazývá interpreter).

Jak to, že nám ho diagonální argument nezakáže? Vysvětlení je v tom, že některé programy se nezastaví. Podrobněji, diagonální úvaha by probíhala takto:

Univerzální program (jednovstupový) označme U. Dostane zápis nulavstupového programu a pracuje jako on. Mírně ho upravme. Budeme chtít, aby vstupy byly zápisy jednovstupových programů a pozměněný univerzální program emuloval

jejich chod na jejich zápis. Proto na začátek přidáme kód, který dosadí vstup p do něho samého – dál tedy postupuje už nulavstupový program. Další kód našeho pozměňovaného univerzálního programu necháme beze změny a na úplný konec přidejme instrukce, které zajistí, že se nějak (jakkoli) změní výsledek výpočtu. Takto pozměněný univerzální program nazvěme V.

Nyní přichází diagonální argument. Spusťme V na V. Spor. Kde je chyba?

Chyba je skryta ve slovech „nějak změní výsledek výpočtu“. Aby to mohl udělat, musí vědět, jaký výsledek je. Když nějaký má, může ho změnit. Ale výpočet se může zacyklit. Ani zacyklit není správné slovo, navozuje dojem, že se něco opakuje. V tom případě bychom náš program mohli vylepšit, že by si zapisoval stavy všech proměnných a číslo instrukce, která je právě na řadě (tzv. konfiguraci), a kdyby zjistil, že se opakovaně ocitá ve stejné konfiguraci, nabyl by jistoty, že se emulovaný program nezastaví, a v tom případě by se zastavil.

Problém je, že výpočet může být nekonečný a nemusí být zacyklený. A nelze to předem zjistit. Kdybychom hledali analogii s oním teologickým příkladem, řekli bychom, že náš bůh je sice všemocný, ale některé věci nikdy nedodělá. Že ví, jak tvořit tak velký kámen, a může tak kdykoli začít činit, ale nikdy ho nedotvoří.

Při tvorbě Gödelova programu i v předchozí falešné argumentaci jsme k dosažení autoreference použili jistou konstrukci, kterou v následující větě explikujeme a dokážeme.

Věta (o rekurzi): Buď f 1-program. Pak na základě jeho

zápisu umíme napsat 0-program a, že f spuštěný na a pracuje stejně, jako pracuje a.

Poznámka:

a) Programu a se často říká pevný bod programu f a větě o rekurzi věta o pevném bodu.

b) V dalším je lhostejné, zda se zajímáme jen o programy bez výstupu nebo i s výstupem.

c) Znění věty nemůžeme uvést ve formě: pak existuje a, že $a = f(a)$. Musí znít, že spuštění a má stejné výsledky jako spuštění f na vstup a. Neboli že spuštění a má stejné výsledky jako spuštění f, do kterého jsme předtím dosadili a.

d) Důkazy uvedeme dva. Oba dávají konkrétní konstrukci, přičemž první je jednodušší. První důkaz lze vymyslet stejnými úvahami, jako už zde byly prezentovány, důkaz je samozřejmě uveden ve vybroušené verzi, takže to nemusí být úplně patrno. Vymyslet druhý důkaz vyžaduje určitý čas, ale vcelku se i k němu dá dostat přímočaře.

Důkaz 1:
Zápis jednovstupového programu f máme k dispozici. Popišme jednovstupový program e. Vstupem e budou jednovstupové programy x. Program e nejprve vezme zápis x a dosadí ho do něho samého. Na tento výsledek pak začne pracovat jako f. Dále předpokládáme, že máme zápis tohoto e. Nyní e dosadíme do e. Skutečně jednou dosadíme, můžeme to udělat ručně nebo k tomu využít tu sadu instrukcí z předchozí věty. Výsledný program je požadovaný program a.

Důkaz 2:
Tentokrát bude e mnohem složitější. Zatímco v prvním důkazu jsme napsali pár řádek pro dosazování vstupu do vstupu, nyní budeme programovat interpreter (=univerzální program). Chceme, aby e pracovalo tak, že vstupní jednovstupový program emuluje, jako by pracoval spuštěn na svůj zápis. To je obrovský kus práce napsat interpreter a to, že nám vždy půjde o emulaci programu na vlastní zápis ji nijak neulehčí. Předpokládejme, že jsme tuto práci vykonali a máme takový jednovstupový e. Nyní budeme chtít další pomocný jednovstupový program v. Ten bude mnohem jednodušší, bude pracovat tak, že vezme vstupní data y, dosadí je do e a na výsledek bude pracovat jako f. Když ho budeme sepisovat vezmeme si k ruce zápis e, napíšeme sadu instrukcí, které zajistí dosazení vstupních dat do e a za tuto sadu opíšeme všechny instrukce f. Nyní (klidně ručně) dosadíme v do e. To je hledaný a.

Poznámka:

a) Obě konstrukce vypadají hrozivě, ale není tomu tak, chce si to podrobně promyslet.

b) Jak již bylo řečeno v obou případech je důkaz skutečně konstruktivní. Z toho plyne, že bychom dokonce mohli napsat programy třeba s názvy PevnyBod1 a PevnyBod2, které by pro zadaný zápis jednovstupového f, vytiskly zápisy a.

V následujícím příkladu se nedržíme omezení na programy bez výstupu. Důvod je estetický, na programech bez výstupu toho není příliš vidět.

Příklad (program, který vytiskne sám sebe):

```
program t(output);
const d=37;
      p=27;
      z=8;
var a:array[11..10+p,1..d] of char;
    r:11..10+p;
    i:1..d;
begin
a[11]:='program␣t(output);␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[12]:='const␣d=37;␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[13]:='␣␣␣␣␣␣p=27;␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[14]:='␣␣␣␣␣␣z=8;␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[15]:='var␣a:array[11..10+p,1..d]␣of␣char;␣␣';
a[16]:='␣␣␣␣r:11..10+p;␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[17]:='␣␣␣␣i:1..d;␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[18]:='begin␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[19]:='for␣r:=11␣to␣10+z␣do␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[20]:='␣begin␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[21]:='␣␣for␣i:=1␣to␣d␣do␣write(a[r,i]);␣␣␣␣';
a[22]:='␣␣writeln␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[23]:='␣end;␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[24]:='for␣r:=11␣to␣10+p␣do␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[25]:='␣begin␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[26]:='␣␣write(''a['',r,'']:='''''');␣␣␣␣␣␣␣␣␣␣␣␣';
a[27]:='␣␣for␣i:=1␣to␣d␣do␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[28]:='␣␣␣if␣a[r,i]='''''''''␣then␣write(''''''''''')␣';
a[29]:='␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else␣write(a[r,i]);';
a[30]:='␣␣writeln(''''''';'')␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[31]:='␣end;␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[32]:='for␣r:=11+z␣to␣10+p␣do␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[33]:='␣begin␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
a[34]:='␣␣for␣i:=1␣to␣d␣do␣write(a[r,i]);␣␣␣␣';
a[35]:='␣␣writeln␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣';
```

```
a[36]:=' end                                 ';
a[37]:='end.                                 ';
for r:=11 to 10+z do
 begin
  for i:=1 to d do write(a[r,i]);
  writeln
 end;
for r:=11 to 10+p do
 begin
  write('a[',r,']:=''');
  for i:=1 to d do
   if a[r,i]='''' then write('''''')
                  else write(a[r,i]);
  writeln(''';')
 end;
for r:=11+z to 10+p do
 begin
  for i:=1 to d do write(a[r,i]);
  writeln
 end
end.
```

Tento program byl získán konstrukcí z prvního důkazu věty o rekurzi (a mírně zkrášlen). Program f je v tomto případě program, který na výstup opisuje vstup. Zkrášlení spočívá například v číslování řádků od 11 místo od 1 (aby mělo číslo řádku vždy stejný počet číslic).

Připomeňme, že program f v Pascalu nemůže prostě jen na výstup posílat symboly ze vstupu, ale musí kontrolovat, jestli symbolem není apostrof, a pokud ano, poslat na výstup dva. Z toho pramení ono kupení se apostrofů. Jestli je něco překvapivé, tak spíš to, že toto kupení neprobíhá do nekonečna,

ale vystačíme s řetězcem nejvíce dvanácti apostrofů za sebou.

Na programu je vidět, jak takové podobné programy a formule vzniklé diagonalizačním dosazením vypadají. Uvnitř v sobě mají celou informaci o sobě.

Je těžké v tuto chvíli nevzpomenout na DNA a její dvojitou šroubovici. Jeden molekulární biolog, se kterým jsem o tom mluvil, to označil za naprosto nesouvisející. R. Penrose na druhou stranu to za náhodu nepovažuje. Těžko říci, rozhodně ale platí, že má-li se nějaký systém (cokoli organismus, stroj, informace) samoreprodukovat musí mít v nějakém smyslu přístup k informaci o svojí konstrukci. Podobně jako fotograf, který se fotí před zrcadlem, jak se právě fotí před zrcadlem ... .

Paralela ale neplatí tak úplně. Fotograf vzhledem ke konečné rychlosti světla zachytí svůj stav o nepatrný okamžik dříve, než probíhá ono zachycování. Mluvím o tom zde proto, že když úlohu „napište program, který vypíše sám sebe“ dostane zadanou programátor inženýr, napíše krátký program pro tisk souboru se jménem `tiskni.exe` a uloží ho pod tímto jménem.

Je prakticky nemožné mu vysvětlit, že zadání nesplnil. Využil při tom totiž spoustu dalších funkcí operačního a souborového systému a jeho výsledek jaksi není „samonosný“, například po překopírování na jiné místo nefunguje (to ovšem inženýr zase vyřeší tím, že nejprve dá prohledat všechny adresáře, aby ho našel).

Jsou to delikátní úvahy. Myslím, že dvojitost šroubovice, zde skutečně nemá tento autoreferenční význam, že příroda v podstatě používá onen inženýrský přístup.

Poznámka (o filozofické interpretaci Gödelovy věty): Filozofové mají obecně tendenci interpretovat Gödelovy věty o neúplnosti jakožto ilustraci nebo i důkaz toho, že lidská mysl má větší schopnost poznání než „suchopárná formální matematika“. I když by bylo možno podobné výlevy přejít bez povšimnutí, možná by stálo za to zamyslet se, jestli na nich přece jen něco není. Jak už bývá u podobných otázek pravidlem, kruciální je správně formulovat otázku. Nalezení odpovědi pak už bývá lehčí částí.

Představme si, že jeden z našich kolegů je skutečne tak suchopárný, že jediné, co ho zajímá a co je ochoten číst, jsou formální teorie. Může to být skutečný kolega ale také to může být software určený pro strojovou kontrolu důkazů matematických vět, jak je to pěstováno například v polském časopise pro formální matamatiku. Co mu předložíme v případě první Gödelovy věty o neúplnosti? Jinými slovy, lze tuto větu formalizovat a pokud ano, tak jak a kde? V dalších kapitolách se pokusím odpovědět.

Až doposud jsme se snažili omezit matematický formalizmus na minimum za cenu leckdy poněkud rozvleklého textu opisujícího něco, co by s použitím formálního jazyka bylo mnohem stručnější. Ukažme si, jak by mohly vypadat formální zápisy. Kvůli tomu se domluvíme se na jistých definicích. Nejprve ale předběžné úvahy.

Kromě predikátu $!(p, k)$ bychom potřebovali také něco jako predikát $\mathsf{Proof}(\varphi, d)$, který by nám vyjadřoval, že řetězec d je důkazem formule φ v naší teorii (φ jakožto textový řetězec je termem, čili na ní můžeme aplikovat predikát). Poznamenejme, že bychom ho na tomto místě mohli nadefinovat

prostředky, které máme k dispozici. Mohli bychom ale postupovat i tak (protože je primitivně rekurzivní), že bychom ho přidali do jazyka podobným postupem jako predikát !, tzn. přijali bychom jako axiomy všechny formule $\mathsf{Proof}(\varphi, d)$, pro které to platí.

Nehrozí zde nebezpečí definice kruhem? Aby se nám totiž nestalo, že ve formuli φ už je predikát Proof použit. Rozmysleme si, že nehrozí. Jakoukoli formuli čteme od nejvnitřnějších závorek směrem ven, a kdykoli narazíme na definovaný pojem, napíšeme místo něj definici.

V našem textu se sice hojně vyskytují autoreferenční zápisy – v tomto případě by to bylo něco $\varphi \equiv \mathsf{Proof}(\varphi, d)$ – ale věta o rekurzi je v podstatě o tom, že se toho dá dosáhnout formálně čistým způsobem, právě bez definice kruhem.

Dalším symbolem, který zavedeme a vztahují se na něj stejné úvahy je symbol Pr. $\mathsf{Pr}(\varphi)$ bude označovat formuli, která říká, že program systematicky hledající důkaz sentence φ uspěje a zastaví se.

Definice:

- Pro 1-program p a jakýkoli textový řetězec d, zápis $p(d)$ značí výsledek dosazení řetězce d do 1-programu p. Je to tedy 0-program.
- Zápis programu, který bude nejprve počítat podle instrukcí programu p a po jeho případném skončení bude pokračovat vykonáváním instrukcí programu q, budeme zapisovat $q\,p$.
- Pro program, který nic dělat nebude, ale hned skončí, použijeme označení konec.
- Pro program, který nic dělat nebude, ale hned skočí do

nekonečné smyčky, použijeme označení ∞.

– Symbol U bude značit tzv. univerzální program pro nulavstupové programy. Tento program je jednovstupový a po jeho spuštění na nulavstupový p pracuje naprosto stejně, jako pracuje p.

– Výsledek spuštění 0-programu p budeme zapisovat jako $[p]$ (zde je nutno podotknout, že to nemusí být definováno, program p nemusí skončit).

– Zápisem $[p]d$ budeme mínit výsledek spuštění jednovstupového programu p na vstup d.

– Při návrhu jednotlivých programů budeme občas užívat podobný obrat jako je tento: „p buď takový, že $[p]x$:='zjisti, jestli se ve vstupu x vyskytuje slovo ahoj, a pokud ano, konec, jinak ∞'“. Tím definujeme p – kdybychom měli dost času a místa mohli bychom ho celý konkrétně sepsat.

Poznámka k definici:

– Podle předchozích úmluv můžeme například napsat $[p]x := x(x)$. p je potom program, který provádí tu operaci dosazení, o které byla řeč v předchozím. Když teď napíšeme $p(p)$, máme tím na mysli, že jsme tento dosazovací program dosadili do něho samého. Celé to je samozřejmě zamotané a matoucí, ale to je vždy, když se mluví o autoreferenci.

– Pro U opět platí, že celá tato definice je jen kvůli šetření času a místa. Program je zcela konkrétní a mohli bychom ho zde na několika (pravděpodobně desítkách) stranách napsat.

– Ve smyslu našich definic platí pro nulavstupový program p triviálně $[\mathsf{U}]p \cong [\mathsf{U}(p)] \cong [p]$. Slovy: spustit U na zápis nu-

lavstupového programu p je to samé jako ho dosadit do U a pak spustit výsledek a je to to samé jako rovnou spustit p. Místo rovnosti jsme zde raději použili znak $\cong$ pro takzvanou podmíněnou rovnost, neboť procesy nemusejí skončit. Jestliže ale skončí jeden, skončí i druhé dva. V dalším nebudeme rozlišovat $\cong$ a $=$ a i v případě, že není jasná zástava budeme užívat znak $=$.

Sepišme nyní znovu větu o rekurzi i oba výše uvedené důkazy s použitím nového značení.

Věta (o rekurzi): Buď f 1-program. Pak na základě jeho zápisu umíme napsat 0-program a, že $[a] = [f]a$

Důkaz 1:
Je dán 1-program f. Napišme e takové, že $[e]x = [f]x(x)$ a $a := e(e)$. Nyní ukažme, že věta platí.
$[a] = [e(e)] = [e]e = [f]e(e) = [f(e(e))] = [f(a)] = [f](a)$

Důkaz 2:
Tentokrát napíšeme e, aby $[e]x = [\mathsf{U}]x(x)$. $[\mathsf{U}]x(x)$ je samozřejmě rovné $[x(x)]$, ale u prvního zápisu více vynikne, že je ve hře univerzální program. Naprogramujeme v, aby $[v]y = [f]e(y)$ a a dostaneme tak, že dosadíme v do e, neboli $a{=}e(v)$. Nyní:
$[a] = [e(v)] = [e]v = [\mathsf{U}]v(v) = [v(v)] = [v]v = [f]e(v) = [f]a$
Konec důkazů.

Vidíme, že naše zápisy se velmi podobají zápisům v λ-kalkulu, což samozřejmě není náhoda.

Pokračujme dále v úvahách o programech. Při jejich konstrukcích můžeme pro pohodlí užívat větu o rekurzi, důležité

však je, že veškeré úvahy jsou konstruktivní a vždy vedou k napsání konkrétního programu.

V našich úvahách budeme často uvažovat a konstruovat programy, které budou hledat důkazy nějakých (obecně jakýchkoli ne jen Σ_1) formulí.

Při tvorbě programů budeme využívat některá makra a konstrukce. Vyjmenujme je explicitně a u každého přidáme poznámku, co to znamená pro jeho zástavu.

Prvním je již zmíněné dosazení dat d (což zrovna tak může být zápis programu) do programu p. Toto makro vždy skončí. Výsledek značíme $p(d)$.

Druhým je rovněž již zmíněné sériové spuštění programů p a q za sebou. Skončí tehdy a jen tehdy, pokud skončí p i q.

Třetím je pro zadaný zápis 0-programu p a číslo n zjistit, zda se p zastaví po n krocích. Toto makro vždy skončí.

Čtvrtým je ověření, zda zadaný textový řetězec je důkazem zadané formule. Rovněž je jisté, že vždy skončí.

Pátým je spuštění programů p a q „paralelně“. To znamená střídavě vykonávat po jedné instrukci z každého programu a jakmile jeden výpočet skončí, zastavit se. Toto makro skončí tehdy a jen tehdy, pokud skončí alespoň jeden z programů.

Šestým je nekonečná smyčka. Toto makro nikdy neskončí.

6 Aplikace – druhá část

Vzpomeňme, že při úvahách o nerozhodnutelnosti zástavy Gödelova programu byly jisté nepříjemnosti při demonstraci toho, že se nám nemůže podařit napsat důkaz o jeho zástavě. Abychom se z předpokladu, že máme důkaz o existenci určitého čísla (v našem případě počtu kroků, po kterém stroj stojí), pohli dopředu, potřebujeme ono číslo tak říkajíc držet v ruce. Uvedli jsme, že toto je mimo jiné obsahem věty známé jako věta o Σ_1 korektnosti. Ale zatím jsme v situaci, že ať se snažíme, jak se snažíme, nemáme žádnou metodu, jak z důkazu o existenci čísla daných vlastností získat alespoň horní odhad pro velikost tohoto čísla. Horní odhad by nám naprosto dostačoval, pomocí něho bychom si už ono číslo opatřit uměli. Že to není náhoda, uvidíme později.

S aparátem, který teď máme k dispozici umíme z této šlamastiky vybřednout. Sice ne tak, že bychom získali onen horní odhad, ale tak, že sepíšeme program, u kterého to nebude potřeba. Navíc se to při těchto úvahách vysloveně nabízí.

Dalším uvažovaným programem bude proto program[2] – označíme ho r a nazveme Rosserovým – který pracuje tak, že paralelně hledá důkaz $!\mathsf{r}$ i $\neg!\mathsf{r}$. V prvním případě následuje nekonečná smyčka, ve druhém konec.

Můžeme si dovolit takto posat program, protože máme větu o rekurzi. Kdybychom se bez ní chtěli obejít, sepsali bychom nejprve program, který toto zjišťování provádí pro vstupní

[2] Mohli bychom ho nazvat sabotér – alibista. Hledá důvody, proč nikdy neskončit, ale raději současně s tím pátrá, zda neexistují důkazy pro jeho skončení (pro případ, že by na ně mohl přijít nadřízený), a jakmile na ně přijde, upadne do nekonečné smyčky, ze které není návratu.

jednavstupový program dosazený do něho samého. Poté tento program dosadíme do něho samého. To byla kontrukce z prvního důkazu věty o rekurzi.

Následující úvaha funguje pro jakýkoli program, který je řešením z věty o rekurzi a nepotřebujeme k ní Σ_1 korektnost.

- Předpokládáme, že máme vytisklý důkaz o nezástavě programu r. Důkaz má určitou délku, řekněme 1126 znaků. Nyní projdeme všechny důkazy kratší než 1126 znaků, jestli mezi nimi náhodou není kratší důkaz nezástavy. Pokud je, vytiskneme si nejkratší. Z této úvahy plyne, že bez újmy na obecnosti můžeme předpokládat, že důkaz je nejkratší možný. Nyní spustíme program r. r postupně prochází všechny důkazy seřazené podle velikosti. Když dojde až k našemu důkazu, zastaví se. Co se může stát před tím?
  - Nalezne kratší důkaz o svojí nezástavě.
  - Nalezne kratší důkaz o svojí zástavě.
  - Nenalezne nic a dospěje až ke zkoumání vytisklého důkazu.

  První případ není možný.
  Ve druhém případě jsme schopni tento důkaz jeho zástavy také nalézt (je kratší než 1126 znaků) a transformovat ho společně s původním důkazem na důkaz sporu v PA.
  Ve třetím případě ho prozkoumá a skončí. Z toho sepíšeme důkaz o jeho zástavě.
  Závěr: jsme schopni transformovat každý důkaz o jeho nezástavě na důkaz sporu v PA.
- Předpokládáme, že máme vytisklý důkaz o jeho zástavě, ten má zase určitou délku, řekněme opět 1126 znaků. Stej-

ným obratem jako v předešlém můžeme zajistit, že je nejkratší možný. Nyní spustíme program r. r postupně prochází všechny důkazy seřazené podle velikosti. Když dojde až k našemu důkazu, skočí do nekonečné smyčky. Co se může stát před tím?

- Nalezne kratší důkaz o svojí zástavě.
- Nalezne kratší důkaz o svojí nezástavě.
- Nenalezne nic dospěje až ke zkoumání vytisklého důkazu.

První případ není možný.
Ve druhém případě jsme schopni tento důkaz o jeho nezástavě také nalézt (je kratší než 1126 znaků) a transformovat ho společně s původním důkazem na důkaz sporu v PA.
Závěr: jsme schopni transformovat každý důkaz o jeho nezástavě na důkaz sporu v PA.

Během dokazování jsme nepotřebovali žádné dodatečné předpoklady, konstrukce funguje vždy.

Shrňme, co jsme dokázali.
Napsali jsme jistý konkrétní program f, konkrétní podoba tohoto f závisí na axiomech PA. Demonstrovali jsme, že pro každý jeho pevný bod a ve smyslu věty o rekurzi, je formule $!a$ nerozhodnutelná. Přesněji, podali jsme jasný konečný návod, jak jakýkoli případný důkaz této formule předělat na důkaz sporu v PA a jak jakýkoli případný důkaz negace této formule předělat na důkaz sporu v PA. Dále jsme sepsali dva takové (z důkazů věty o rekurzi) konkrétní pevné body. Čemu jsme se nevěnovali je zkoumání případné ekvivalence těchto dvou (a obecně všech) formulí získaných z pevných bodů. Tomu je věnován článek [9].

Uvažme nyní situaci, že by místo PA stála nějaká jiná teorie. Na jakém místě může případně naše konstrukce nebo argumentace selhat? Výše uvedené důkazy bez problémů projdou i po nahrazení pojmu PA jakoukoli jinou teorií. Aby však fungovala celá konstrukce, musíme být schopni onoho překladu formule $!p$ do jazyka oné teorie. Jinými slovy, teorie musí být dostatečně silná, aby v ní šly namodelovat programy a formule vyjadřující jejich zástavu.

Do naší galerie takto opatřitelných programů se přirozeným způsobem nabízejí ještě čtyři. První dva uvedeme jen pro pořádek, neboť situace je u nich jasná.

První hledá důkaz, že se zastaví, a když ho najde, skočí do nekonečné smyčky.

Druhý hledá důkaz, že se nezastaví, a když ho najde, skočí do nekonečné smyčky.

V obou případech si dovoluji tvrdit, že neskončí a že je lehké to dokázat.

Jako třetí budeme zkoumat program[3] – h a nazveme Henkinovým – který hledá důkaz, že se zastaví, a když ho najde, tak skončí. Dostaneme ho zase stejným způsobem z věty o rekurzi.

Zde budeme obšírnější. Když je otázka položena takhle, je vlastně jasné, jaký konkrétní program máme na mysli. V podstatě tím říkáme, že nám jde o program z prvního důkazu věty o rekurzi.

Předchozí programy jsme konstruovali přímo s cílem najít

[3]Pro něj se hodí název *parazit* nebo *darmošlap*. Nědělá nic jiného, než že hledá důkaz, že jednou skončí.

nerozhodnutelné tvrzení. Ke splnění tohoto hlavního cíle nám stačil v obou případech jeden exemplář. Přesněji řečeno po konstrukci a úvahách s Gödelovým programem jsme právě kvůli nespokojenosti s naplněním cíle pokračovali v hledání, které jsme úspěšně završili Rosserovou konstrukcí (Rosserův program je vlastně zvláštním případem Gödelova).

V tuto chvíli budeme raději a obšírnější.

Napsali jsme f. To má jako vstup zápis nulavstupového programu a hledá důkaz, že se tento program zastaví. Po jeho případném nalezení skončí. Přesně takhle vypadá f. Nyní nás zajímají programy, o kterých můžeme dokázat, že skončí právě tehdy, když skončí f spuštěný na ně. Formálně, že

$$[f]\mathsf{h} = [\mathsf{h}] \qquad (*)$$

Že takové programy existují, se dovídáme z věty o rekurzi. Pro nejméně jeden z nich se hodí ono jméno darmošlap, protože skutečně nedělá nic jiného. Je ale mezi nimi i spousta jiných programů. Například program, který nejprve počítá číslo π na sto desetinných míst a pak teprve začne hledat důkaz svojí zástavy.

Ptáme se vlastně na několik otázek:

1) Existuje takový h splňující $(*)$, že se zastaví?
2) Zastaví se h z prvního důkazu věty o rekurzi?
3) Zastaví se h z druhého důkazu?
4) Zastaví se každý h, který splňuje $(*)$?

První otázka je jen pro pořádek, dávno víme, že ano.

Představme si nyní, že se píše rok 1952 a právě jsme dočetli onen Henkinův článek, kde tuto otázku položil. Já sám si to umím představit celkem lehko, neboť při prvním studiu této

problematiky na MFF jsme končili u Gödelovy formule a až do fáze, ve které jsme teď, jsem se přes Rosserův program dostal vlastními silami.

Odpovědět na celý seznam těchto otázek jsem však nedokázal. Naopak se mi zdálo být intuitivně jasné, že se takový program nezastaví. Tipoval jsem mezi dvěma možnostmi. Že se buď podaří dokázat, že se program nezastaví, anebo bude problém nerozhodnutelný (což by ve standardním modelu znamenalo, že se nezastaví). Jaké bylo posléze moje překvapení během studia na FF, když jsem shlédl Löbův důkaz! Opravdový šok.

Když jsem se tenkrát zamýšlel nad tímto problémem, dospěl jsem k názoru, že se mi pro konkrétní program z prvního důkazu věty o rekurzi podaří alespoň dokázat, že je pro tuto úlohu úplný. Že totiž problém zastavení jakéhokoli jiného pevného bodu na něj bude převeditelný.

Samozřejmě že ve světle Löbova důkazu tato skutečnost postrádá význam, podobně jako by ztratila význam Cookova věta o úplnosti problému SAT, kdyby někdo dokázal, že $P = NP$. I dnes zpětně si ale stále myslím, že by důkaz nebyl těžký. Pouze už nemá smysl se jím zabývat.

Vyzbrojeni Löbovou úvahou jsme schopni na všechny otázky ze seznamu odpovědět kladně. V původním Löbově důkazu se pracuje nejen se Σ_1 formulemi, což znamená, že se nedá přeložit do našeho způsobu mluvení o zástavách či nezástavách programů, což je také velmi překvapivé. Je však možné jeho ideu modifikovat a pak to funguje. Otázkou je, nakolik bude tato přeformulace přirozená.

7 Löbův důkaz

Tak jako se pravděpodobně Gödel nechal inspirovat lhářovým paradoxem „tento výrok je nepravdivý“ dá se říci, že Löbův důkaz se inspiruje výrokem „je-li tento výrok pravdivý, jsem papež“. Pokud by byl nepravdivý, je nepravdivý předpoklad tedy je pravdivý. Spor. Dokázali jsme, že je pravdivý, a protože je pravdivý a je pravdivý i jeho předpoklad, platí i závěr, jsem papež.

Zatímco u lhářova parodoxu nenarazil Gödel při nahrazování pojmu pravdivosti pojmem dokazatelností na žádné komplikace, při přeformulaci tohoto paradoxu zjistíme, že funguje jen pro výroky, pro něž platí $\mathsf{Pr}(\varphi)\rightarrow\varphi$. Ostatně kdyby tato „komplikace“ nenastala, bylo by to špatné, znamenalo by to, že je PA sporná.

Zopakujme nejprve Löbův důkaz. Vstupem programu f je zápis nulavstupového programu. f hledá důkaz, že se tento program zastaví a po jeho případném nalezení končí. Nějaký pevný bod tohoto programu f máme označen h. Aritmetickou formuli, která vyjadřuje zástavu h a kterou dostaneme konstrukcemi popsanými v předchozím textu, označíme σ.

Je zřejmé, že σ splňuje $\sigma \equiv \mathsf{Pr}(\sigma)$. Máme na mysli, že je zřejmé, že je to dokazatelné v PA. Naopak, z každé formule, která splňuje tento vztah, jsme schopni zkonstruovat program – pevný bod programu f.

Pro následující Löbův obrat nemáme v našem textu vybudovaný dostatečný formální aparát. Použije tzv. Gödelovo diagonální lemma (my ho díky větě o rekurzi vlastně máme jen pro Σ_1 formule), aby dostal τ takovou, že $\tau \equiv (\mathsf{Pr}\ \tau \rightarrow \sigma)$.

Postupu je třeba rozumět tak, že τ je zcela konkrétní (samozřejmě je v něm nějak zapracována σ) a ekvivalenci umíme dokázat v PA.

Nejprve si formulujme pro Pr takzvané Löbovy podmínky dokazatelnosti. Připomeňme, že v našem textu $\mathsf{Pr}(\varphi)$ znamená, že se zastaví program hledající důkaz φ.

1) Jestliže dokážeme φ, dokážeme i $\mathsf{Pr}(\varphi)$.
   Tomu je potřeba rozumět konstruktivně. Máme algoritmus jak důkaz sentence předělat na důkaz její dokazatelnosti.
2) Jestliže $\mathsf{Pr}(\varphi)$ a $\mathsf{Pr}(\varphi \to \psi)$, pak $\mathsf{Pr}(\psi)$.
3) Jestliže $\mathsf{Pr}(\varphi)$, pak $\mathsf{Pr}(\mathsf{Pr}(\varphi))$.

Podmínky 2) a 3) jsou sice metapodmínky ale jen v tom smyslu, že jsou to schemata. Pro každé konkrétní formule φ a ψ dokážeme obě tvrzení. Nemluví se v nich nic o tom, že něco je dokazatelné, pokud něco jiného je dokazatelné.

Rozmyslet si, že tomu tak skutečně je, není v našem jazyce obtížné, jen je potřeba dát pozor a nepoplést hladinu matematickou s hladinou metamatematickou.

Nyní Löb překvapivě dokáže τ:

Předpokládá $\mathsf{Pr}\ \tau$.
Z toho hned dostane $\mathsf{Pr}(\mathsf{Pr}\ \tau)$ a $\mathsf{Pr}(\mathsf{Pr}\ \tau \to \sigma)$.
Díky druhé podmínce $\mathsf{Pr}(\mathsf{Pr}\ \tau) \to \mathsf{Pr}(\sigma)$.
Dohromady tedy $\mathsf{Pr}(\sigma)$ a protože je $\sigma \equiv \mathsf{Pr}(\sigma)$ je hotov.

Otázkou pro nás nyní je, jestli jsme schopni totéž dokázat s použitím našeho formalismu. Není totiž nikterak jasné, jestli formule τ, kterou jsme dostali z diagonálního lemmatu, je Σ_1. Spíše to vypadá, že není. Pokud není Σ_1, znamená to, že

nejsme schopni napsat program, který by jí odpovídal.

Nebudeme zkoumat, zda je či není Σ_1, místo toho předvedeme důkaz, který použije jinou Σ_1 formuli.

Nejprve napišme τ takovou, že
$\tau \equiv \mathsf{Pr}(\mathsf{Pr}(\tau) \rightarrow \sigma)$[4]; to můžeme z věty o rekurzi
Nyní τ dokážeme; k tomu stačí dokázat $\mathsf{Pr}(\tau) \rightarrow \sigma$. Jakmile totiž toto budeme mít, budeme mít i $\mathsf{Pr}(\mathsf{Pr}(\tau) \rightarrow \sigma)$.
Předpokládáme $\mathsf{Pr}(\tau)$. Chceme dokázat σ.
Z „definice“ τ: $\mathsf{Pr}(\mathsf{Pr}(\mathsf{Pr}(\tau) \rightarrow \sigma))$.
Z druhé podmínky: $\mathsf{PrPrPr}(\tau) \rightarrow \mathsf{PrPr}(\sigma)$
Z $\mathsf{Pr}(\tau)$ dostaneme $\mathsf{PrPr}(\tau)$ a z toho $\mathsf{PrPrPr}(\tau)$ díky 1).
Takže $\mathsf{PrPr}(\sigma)$ a díky tomu, že je $\mathsf{Pr}(\sigma) \equiv \sigma$, dostáváme σ.
Teď jsme v situaci, že jsme dokázali τ.
Umíme tedy i dokázat $\mathsf{Pr}(\tau)$ a $\mathsf{PrPr}(\tau)$.
Víme už, že platí τ, což znamená že i $\mathsf{PrPr}(\tau) \rightarrow \mathsf{Pr}(\sigma)$.
Dostaváme $\mathsf{Pr}(\sigma)$ a hned σ.

Přeříkat tento důkaz pro odpovídající program z prvního důkazu věty o rekurzi lze, neodvažuji se ho zde však uvést, neboť se při něm skutečně láme jazyk. Pravděpodobně by na něj takto asi těžko jdo přišel.

[4] Naproti tomu s použitím formule $\tau \equiv \mathsf{Pr}(\tau \rightarrow \sigma)$, kterou navrhuje použít Švejdar ve cvičení 5.3.2 knihy [21] se mi důkaz provést nepodařilo.

8 Poslední na řadě

Posledním programem, který zcela samozřejmě zbývá ke zkoumání je program, jenž paralelně hledá důkazy své zástavy či nezástavy a pak se zastaví či zacyklí. Vyhradíme si pro něj písmeno x.

Co s tímto programem?

Hned na začátek předesílám, že odpověď se mi nepodařilo nalézt. Myslím tím jednak vlastní bádání, jednak literaturu. Přijde mi zvláštní, že by se tímto problémem nikdo nezabýval, když jeho formulace je tak přirozená, nicméně jsem o něm nic nenašel.

Nejprve uveďme všechno, co víme.

Program f tentokrát pro zadaný nulavstupový program hledá paralelně jednak důkaz, že se p zastaví, a jakmile ho najde, končí, a jednak důkaz, že se p nezastaví, a v tom případě skáče do nekonečné smyčky.

Písmenem x označujeme každý pevný bod programu f, neboli každé x, které splňuje

$$[f]\mathsf{x} = [\mathsf{x}] \qquad (**)$$

V aritmetickém jazyce je to ekvivalentní problému, zda je formule ξ, která splňuje autoreferenční rovnici

$$\xi \equiv \exists d\ \mathsf{Proof}(\xi, d)\ \forall e\ \mathsf{Proof}(\neg\xi, e) \rightarrow e > d \qquad (***)$$

dokazatelná.

Je zde opět ten samý seznam otázek jako u Henkinova programu.

1) Existuje takový x?
2) Zastaví se x z prvního důkazu věty o rekurzi?
3) Zastaví se x z druhého důkazu?
4) Zastaví se každý x, který splňuje $(**)$?

Odpověď na 1) je triviálně ano.

Na tomto místě mi nezbývá než slovy uvést, co si myslím a jak jsem pokročil v hledání odpovědí. Obě formulace problému v řeči programů i aritmetických formulí jsou ekvivalentní, budeme mluvit jen o programech.

První moje hypotéza je, že čtvrtá a tedy i třetí otázka se zodpoví, poté co se nám podaří kladně odpovědět na druhou otázku.

Mám tím na mysli, že pevný bod x programu f, získaný z prvního důkazu věty o rekurzi je pro tento problém úplný.

V řešení jsem nijak nepokročil, čas jsem věnoval právě řešení druhého problému, že pokud získáme program x druhým důkazem věty o rekurzi, zastaví se.

Problém se zdá být jednoduchý, neboť při prvním pohledu se zdá, že program x nedělá v podstatě nic jiného než Henkinův program h až na to, že to raději ještě na závěr pojistí probírkou všech kratších důkazů, jestli mezi nimi náhodou není důkaz negace, což my víme (díky předpokladu o bezespornosti PA), že není.

Při formalizaci však narazíme na problémy, neboť takhle jednoduše to formalizovat nelze.

Nejnadějnější se mi zatím jeví rozdělit program f na dva programy a a b, které budou pracovat v sérii za sebou s celkovým výsledkem stejným jako f.

Program a se pro zadaný program p snaží nalézt důkaz, že $p(p)$ skončí; v případě jeho nalezení končí. Výstupem tentokrát nebude jen to, že skončil, ale i délka onoho nalezeného důkazu.

b pak probere všechny důkazy kratší než ona délka, jestli mezi nimi není důkaz toho, že $p(p)$ neskončí.

Teď máme rozhodnout, zda $[ba]ab$ (možná ne úplně přesně zapsáno, ale smysl je jasný) skončí.

To je první náčrt idey. Takto sama o sobě nefunguje.

Další vylepšení je, že když programy a nebo b narazí na zkoumání programu složeného ze dvou sériově spojených programů (to jsme schopni snadno zajistit, oddělíme je nějakým komentářem), nejprve hledají důkaz zastavení prvního a pak teprve druhého.

Tím bychom možná mohli použít něco jako „roznásobení“ a převést otázku na čtyři. Jestli se zastaví $[a]a$, $[a]b$, $[b]a$ i $[b]b$.

$[a]a$ vypadá, že je ekvivalentní Henkinovu programu, zastavení b bychom mohli dostat z předpokladu bezespornosti PA.

Nápad jsem bohužel dostal na poslední chvíli, takže zatím nevím, jestli bude fungovat.

Ještě se zmiňme, jaké jsou teoretické jiné možnosti.

V případě, že se nepodaří dokázat, že se program x zastaví, případně nebude pro pro problém úplný, vyvstává legitimní otázka, kterou je stručnější formulovat v aritmetickém jazyce:

Jsou všechna řešení ξ výše zmíněné autoreferenční rovnice $(***)$ ekvivalentní?

Jedno triviální řešení máme, je jím formule $0 = 0$. V případě

„duálního“ programu, tj. Rosserova, například platí, že různá řešení jemu odpovídající autoreferenční rovnice

$$\xi \equiv \exists d\ \mathsf{Proof}(\neg\xi, d)\ \forall e\ \mathsf{Proof}(\xi, e) \rightarrow d < e$$

nemusejí být ekvivalentní. O tom pojednává článek [9].

Jěště zmínka k odpovědi paní učitelky a s ní související věty o Σ_1 korektnosti.

Zmínili jsme, že je možná situace, že se nám podaří dokázat existenci důkazu nejaké sentence φ aniž bychom drželi v ruce samotný důkaz. Neexistuje obecná metoda, jak tento důkaz sentence $\mathsf{Pr}(\varphi)$ transformovat na důkaz φ?

Ukažme si, že neexistuje – respektive k jakým závěrům by to vedlo. Kdyby existovala, bylo by možno na jejím základě sepsat program, který by to dělal. Takže by platilo: jestliže pro libovolnou φ dokážeme $\mathsf{Pr}(\varphi)$, dokážeme i φ. Za předpokladu existence algoritmu, by předešlý řádek šlo formalizovat uvnitř PA, takže bychom v PA dokázali $\mathsf{PrPr}(\varphi)\rightarrow\mathsf{Pr}(\varphi)$. Podle Löbova výsledku by to znamenalo, že jsme schopni dokázat $\mathsf{Pr}(\varphi)$ pro jakékoli φ. Byli bychom tedy schopni dokázat existenci sporu – což je, jak víme z druhé Gödelovy věty sice bezesporné, ale není to nic, co bychom přijali s lehkým srdcem.

Tak jako je plné znění Gödelovy věty velmi opatrné, že totiž za předpokladu bezespornosti PA se její bezespornost nedá dokázat uvnitř ní samé, je i toto znění opatrné: Za předpokladu, že neumíme dokázat existenci formálního sporu, neexistuje ona zmíněná metoda.

A pořád nevím, měla paní učitelka pravdu nebo ne?

9 Reflexe ve společenských vědách

Interpretujme nyní volně predikát dokazatelnosti $\mathsf{Pr}(\varphi)$ jako „věřím, že φ“, případně „je věřeno, že φ“. Protože nemám důvod zpochybňovat seriózní medicínské výzkumy, věřím závěrům týkajícím se tzv. placebo efektu. Věřím, že když pacient věří, že mu neúčinná látka pomůže, skutečně mu pomůže.[5]

Nyní jsem v situaci, že mi homeopat takovou látku předepsal. Pomůže mi nebo ne? Překvapivá odpověď zní, že ano. Argumentace probíhá podobně jako Löbův důkaz.

Jedná se o jakoby nějaké sebenaplňující se proroctví. Člověku vytane na mysli Oidipův příběh. Protože znal věštbu, učinil takové kroky, které nakonec vedly k jejímu naplnění.

V poslední době se s variacemi na podobné téme roztrhl pytel. Harry Potter, Eragon i další fantasy literatura ho obsahuje téměř povinně.

Já jsem se bohužel s touto konstrukcí poprvé nesetkal ani ve fantasy literatuře ani v řeckých tragediích ani při studiu logiky dokazatelnosti ale při výuce občanské nauky v komunistickém podání.

Marx a Engels totiž nezvratně a přísně vědeckými metodami dokázali směřování společnosti ke komunismu. A my, kteří máme možnost jejich učení studovat, ho díky tomuto jejich objevu vybudujeme, čímž dojde k potvrzení jejich výsledku.

Ptát se tehdy, zda-li by došlo k vybudování komunismu i bez toho, aby byla objevena jeho nevyhnutelnost, se jaksi nehodilo. A možná, že odpověď by byla, že objev byl z podobného

[5] Tomuto tématu se žertovnou formou věnuje Smullyan v [18].

důvodu rovněž nevyhnutelný. Logická stavba argumentů by byla pravděpodobně poněkud slabší než v hodině matematiky.

Vůbec se dá obecně říci, že v základech většiny ideologií, které mají tendenci se šířit, je zabudován tento princip jakéhosi sebepotvrzení.

Tak jako v moderní fyzice začal hrát začátkem 20. století neopominutelnou roli pozorovatel, ve společenských vědách začal hrát podobně důležitou roli řekl bych „očekávatel“. Vezměme si například zcela čerstvou zprávu z ekonomického zpravodajství agentury Reuters.

„Index spotřebitelské důvěry ve Spojených státech poklesl ve druhém čtvrtletí o pět procentních bodů.“

Důsledkem této zprávy je samozřejmě jeho další pokles. Kvantitativně tyto systémy s pozitivní zpětnou vazbou studuje teorie dynamických systémů tzv. teorie chaosu nicméně kvalitativní základ je podle mého názoru v tomto autoreferenčním principu. U teorii chaosu bude ještě zmínka.

Dalším příkladem jsou volby a s tím související průzkumy veřejného mínění. Mnohokrát bylo povrzeno, že voliči mají tendenci spíše volit stranu s větší nadějí na vítězství, což vedlo k zajímavému pragmatickému opatření zakázat několik dnů před volbami zveřejňování výsledků aktuálních průzkumů.

Nemarkantněji a nejzajímavěji se tento princip projevuje na trzích cenných papírů a hlavně jejich derivátů. Derivátem je jakýkoli cenný papír, jehož držení má za následek závazek pro držitele ke koupi nebo prodeji podkladového aktiva za aktuální tržní cenu někdy v přesně určené budoucnosti. Uvádí

se, že objem tzv. spekulativních obchodů je přes 98% z celkově otevřených kontraktů. To znamená, že pouhá dvě procenta těchto obchodů vedou ke skutečné fyzické realizaci obchodu.

Představme si, že jsme takovýmto spekulantem a kontrakt, který jsme otevřeli má termín fyzické realizace dejme tomu za pět let. Pro konkrétní představu ať je to například kontrakt na prodej pomerančového koncentrátu. Jestliže proběhne médii zpráva o šířící se plísňové chorobě citrusových stromů, jistě lze očekávat okamžité zvýšení ceny. Avšak hlavním důvodem zvýšení ceny je všeobecné očekávání zvýšení ceny. Je zde vidět jasná autoreference.

Co z toho konkrétně vyplývá. Za prvé to je paradoxní jev, že aby člověk vydělal, musí jít s davem. Většinou je člověk zvyklý, že k úspěchu vede volba jiné cesty než jdou ostatní. Zde tomu tak není. Dále - a to už je spíše otázka pro teorii dynamických systémů – lze očekávat rychlé výkyvy. Pokud je důvodem růstu cen nemovitostí především velké množství spekulativních nákupů za účelem výnosu z rostoucích cen, je možno očekávat, že po snížení nebo zastavení růstu opadne zájem kupovat, což dále sníží růst nebo dokonce cenu atd.

Uvažování o programech, o jejich zastavování s případnou aplikací diagonalizace je podle mého názoru zvláště vhodné k filozofickému zkoumání pojmu svobodná vůle. Je zde onen starý rozpor mezi determinismem, podle kterého se vše řídí deterministickými fyzikálními zákony, a svobodnou vůlí, která se rozhoduje, co a kdy udělá, a budoucnost ovlivňuje.

Filozofové velmi často učiní závěr, že z toho vyplývá jasný principelní rozdíl mezi živou a neživou hmotou. Že totiž ne-

živá hmota se řídí deterministickými fyzikálními zákony,[6] zatímco na živé bytosti obdařené svobodnou vůlí se determinismus nevztahuje. A jejich argument je (ne)překvapivě diagonální.

Kdyby bylo vše jasné, svobodná vůle by se nemohla svobodně rozhodovat, co udělá.

Já osobně považuji za naprosto jasné, že žádný principelní rozdíl mezi živými bytostmi a neživou hmotou neexistuje. Protože zde o tom nehodlám polemizovat, přijměme to pro zbytek úvahy jako předpoklad. Chci ukázat, že i za tohoto předpokladu není determinismus a existence svobodné vůle v rozporu.

Programy se při svých výpočtech rozhodují podle momentálního stavu proměnných a podle podmínek, které právě provádějí.

Celý proces je k nerozeznání od fungování svobodné vůle.

Přesto je jejich chování a výpočet naprosto deterministický. Pointa věci je podle mého názoru v tom, že může být velký rozdíl mezi tím, když řeknu „proces je deterministický“ a „proces je rozhodnutelný (spočitatelný)“.

Jedna konkrétní úvaha, na které to snad bude lépe vidět. Představme si, že nám někdo bude tvrdit, že napsal program - věštce. Sice ne tak úplně, ne až do věčnosti (neumí rozhodnout, zda se program vůbec kdy zastaví), ale pro zadaný nulavstupový program p umí zjistit, jestli bude po tisíci krocích stát. A po úpravách by tam klidně mohlo stát jakékoli

[6] Asi bych měl zdůraznit, že do toho vůbec nehodlám zatahovat moderní fyziku. Žádné různosti historií a vlnové funkce neuvažujeme.

číslo. My mu řekneme „Ale to není nic obtížného, prostě stačí si ten program pustit, počkat tisíc kroků a je to.“

„Ne, ne“, řekne on. „Já to pro každý program zjistím rychleji, než za tisíc kroků!“

Nemůže mít pravdu. Abychom ho přesvědčili, vezmeme jeho program v a upravíme ho. Pokud pro zadaný program p zjistí, že skončí za méně než tisíc kroků, necháme ho vykonat dva tisíce jalových kroků (jedna řádka programu navíc s cyklem). Jinak ho necháme nezměněn.

Argument je zase stejný: Spusťme upravený v. Buď skončí do tisíce kroků nebo ne. Pokud ano, přijde na to podle předpokladu sice dřív než za tisíc kroků, ale spadne do té smyčky. Takže neskončí dřív než za tisíc kroků. Podle předpokladu by na to ale měl přijít a skončit. Spor, dotyčný nám o vlastnostech přineseného programu lhal.

Z tohoto pohledu by nemělo překvapit, že existují programy (funkce) nesložité zápisem ale složitých výsledků. To se ovšem dá očekávat jen pokud bude docházet k opakované iteraci výpočtu na průběžný výsledek. Argument je stejný jako pár řádků výše.

Teorie rekurze a věty o nerozhodnutelnosti vznikly v první třetině 20. století. Z tohoto pohledu je překvapivé, že teorie chaosu vznikla až o více než třicet let později.

Co z toho všeho plyne?

To jestli je svět deterministický nebo ne, nemůže mít žádný praktický dopad.

Budoucnost se předpovědět nedá a to ani v krátkém výhledu.

Budoucnost je otevřená.

Reference

[1] B. Balcar and P. Štěpánek. *Teorie množin.* Academia, Praha, 1986.

[2] J. Barwise. An introduction to first-order logic. In *Handbook of Mathematical Logic*, chapter A.1, pages 5–46.

[3] G. Boolos. *The Logic of Provability.* Cambridge University Press, 1993.

[4] G. J. Chaitin. *The Unknowable.* Springer-Verlag, Singapore, 1999.

[5] A. Church. A note on the Entscheidungsproblem. *J. Symbolic Logic*, 1:40–41, 1930.

[6] A. Church. An unsolvable problem of elementary number theory. *Amer. J. Math.*, 58:345–363, 1930.

[7] O. Demuth, R. Kryl, and A. Kučera. Teorie algoritmů. Lecture notes, Faculty of Mathematics and Physics, Charles University, 1989.

[8] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 37:349–360, 1931.

[9] D. Guaspari and R. M. Solovay. Rosser sentences. *Annals of Math. Logic*, 16:81–99, 1979.

[10] P. Hájek and P. Pudlák. *Metamathematics of First Order Arithmetic.* Springer, 1993.

[11] L. Henkin. A problem concerning provability. *J. Symbolic Logic*, 17:160, 1952.

[12] S. C. Kleene. *Introduction to Metamathematics.* D. van Nostrand, 1952.

[13] M. H. Löb. Solution of a problem of Leon Henkin. *J. Symbolic Logic*, 20:115–118, 1955.

[14] I. L. Novak. Models of consistent systems. *Fund. Math.*, 37:87–110, 1950.

[15] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, New York, 1967.

[16] J. B. Rosser. Extensions of some theorems of Gödel and Church. *J. Symbolic Logic*, 1:87–91, 1936.

[17] J. R. Shoenfield. A relative consistency proof. *J. Symbolic Logic*, 19:21–28, 1954.

[18] R. Smullyan. *Na věky nerozhodnuto.* Academia, Praha, 2003.

[19] A. Sochor. *Klasická matematická logika.* Karolinum, Praha, 2001.

[20] A. Sochor. *Metamatematika teorií množin.* Karolinum, Praha, 2005.

[21] V. Švejdar. *Logika: neúplnost, složitost a nutnost (Logic: Incompleteness, Complexity, and Necessity).* Academia, Praha, 2002.